Chapter 1

Analysis of emerging features of C# language towards mutation testing

Anna DEREZIŃSKA

1. INTRODUCTION

Evolution of programming languages brings emerging programming features aimed at improving programming efficiency and dependability. All programming mechanisms should be taken into account in development of high quality test suites. Tests case generation and qualification can be supported by a mutation approach.

Mutation technique inserts simple changes into a program under test. Faults injected into a source code are defined by mutation operators. Using these operators, many mutated programs, so-called mutants, are generated in a systematic way. Evaluating the adequacy of a test suite, mutants are run against test cases from the suite. A mutant is killed when at least one test case causes it to fail. Mutants that generate exactly the same output as the original program cannot be killed by any test and are called equivalent mutants. Ability to kill mutants is a criterion for generation and selection of test cases.

Standard (traditional) mutation operators are dealing with mistakes common to all general purpose languages, e.g. misusing of relational, logical, arithmetical operators, of variables, constants, etc. [28]. Apart from these operators used in functions of structural programming languages or methods of object-oriented ones, other specialized operators were developed, like intra- and inter-class mutation operators dealing with object-oriented features for Java [13,17] or C# [4-6], mutations of SQL queries [27], concurrent constructs [2] and others. A survey of mutation testing techniques and problems concerning different programming and specification languages can be found in [12].

A basic core of programming languages is usual stable, but a whole language can evolve gaining new features. They can correspond to new possibilities of architecture and operating systems or try to make a programmers work more effective and reliable. The continuous evolution is especially characteristic for C# language. With the release of Visual Studio 2005 the C# language has been updated to version 2.0. A new version 3.0 of the language and its compiler is available in VS 2008 [21,26]. In both cases several new features and enhancements were introduced.

1 Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland, A.Derezinska@ii.pw.edu.pl

This work was supported by the Polish Ministry of Science and Higher Education under grant 4297/B/T02/2007/33.
In this chapter advanced features of C# language are analyzed in the context of possible mutation operators. The main question is, whether a construction can be misused in such a way that the error is not detected at compile time and can be generalized as a mutation operator. Such a mutation operator could be used for the development and evaluation of test cases that would be able to detect errors in this programming construction. Two kinds of mutations are considered, direct changes of C# source code and mutations reflecting changes of C# code introduced into Intermediate Language of .NET.

In the next part of the chapter the background of mutation testing of C# programs will be briefly presented. Next an overview of new C# programming structures and possibility of potential errors and corresponding mutation operators will be analyzed. Discussion about performed experiments and some conclusions finish the chapter.

2. BACKGROUND - MUTATION OF C# PROGRAMS

Object-oriented (O-O) features were primarily studied for Java programs, for which object-oriented mutation operators were proposed [13,17]. The operators were implemented in MuJava [18] and other tools [12] and evaluated in experiments [15,16,24].

Object-oriented operators of Java were adopted for C# and extended with operators specialized to this language, giving a set of about forty mutation operators [5,6]. Their detailed specification was given as transformation with pre- and post-conditions [4,5].

A research in [1] mentioned object-oriented features of C# but concentrated on algorithms for optimization of test cases selection. They referred to standard mutation operators (LOR, NOR, ROR), perturbation on values of constant and variables, an exception and two O-O operators (MCR, RFI). The O-O operators were not studied in detail.

Mutation operators were also developed for specific application domains. In [2] a set of operators for multithreaded constructions in Java programs was proposed. The similar approach could be adopted for development of concurrent application in C#.

Mutation of net applications using ASP.NET (Active Server Pages) was studied in [19]. It took into account a subset of standard and object-oriented operators of .NET languages, like C#, and proposed operators specific for ASP.NET.

Mutation of SQL queries can be realized using mutation operators proposed in [27] and implemented in the SQLMutation tool. They were evaluated also in some experiments on queries taken from industrial applications [7]. Such operators were also used for Java programs interacting with a database via Java Database Connectivity API [29]. They could be adopted for the LINQ notation introduced in C# 3.0.

Simple standard mutation operators for C# language are supported in Nester [22], an analogous tool to Jester developed for Java programs. The basic mutations refer to changes of constants (e.g. 0 to 1), Boolean values to an opposite ones, conditions “if(“ to “if(true ||”, etc. The improved version of Nester makes only one compilation run for all mutants. Afterwards, it is decides during test execution which mutant should run.

The first tool supporting selected object-oriented mutations of C# was the CREAM system [9,10]. It uses a parser-based approach and introduces changes to a C# source
code according to identified places in a program syntax tree. Object-oriented mutation operators are applied after verification of appropriate correctness conditions in order to generate valid mutants, i.e. mutants that compile properly. An enhanced CREAM2 supports 13 object-oriented operators and 5 traditional mutation operators [11].

C# programs, like other programs that run on the .NET platform, are transformed into assemblies consisting of metadata and managed code [3]. Managed code includes body of methods written in an intermediate form called Common Intermediate Language (CIL) [25]. It is a machine-level, stack-based language. It includes instructions for creating, initializing and manipulating object types. It also supports array manipulation and exception handling. If mutating changes are introduced directly into CIL, a duration time of mutants generation can be reduced as compilation of mutants is not necessary.

Several simple changes reflecting standard mutations at the high-language level can be implemented in the Intermediate Language in a straightforward manner via pattern matching [20]. It is not so obvious for the object-oriented and other complex mutations. Mapping of several object-oriented mutation operators of C# program into changes introduced to CIL was presented in [8,14]. Using this approach, six selected object-oriented operators were implemented in a prototype version of the ILMutator tool.

3. OVERVIEW OF ADVANCED C# FEATURES IN THE RELATION TO MUTATION OPERATORS

The C# language was enhanced with several new programming constructions in versions 2.0 and 3.0 [21,26]. They are discussed in terms of a potential source of programmer’s errors and consequently an aim of test cases able to detect such programming flaws. These new features are analyzed whether they can be used in appropriate mutation operators and are summarized in Tab. 1.

The primarily analysis refers to mutations that are introduced directly into C# source code. Similarly to object-oriented operators and other advanced operators of C# an operator requires not only a simple code change but often certain correctness conditions should be checked in order to generate a valid mutant [4,10,11].

Next, a necessary condition of considering the mutation in the Intermediate Language is discussed, if appropriate. Many of new constructions of C# 3.0 are aimed at simplification of programmers’ tasks. They do not extend functionality of the language but allow shortcuts for known programming constructions. At compile time both statements, shortened and explicit ones, are converted to the same corresponding constructions in CIL. Therefore they are not visible, cannot be referred to and can be not considered for mutation introduced at this level. They are denoted in the last column of Tab. 1.

Generics

Generic types were added to C# 2.0 to achieve a high level of code reuse. A generic type definition is a class, structure, or interface declaration that functions as a template, with placeholders for the types that it can contain or use. Several constrains can be associated with parameters of generics. They specify whether a given type is supposed to be
a value type, a reference, is of type or inherits from a class, implements an interface, has
the same type as another parameter or has a nonparametric constructor. Constraints are
checked at compile time and prevent an incorrect usage of a given type. The following
mutation operators were proposed for the generic types.

The GCD operator (Generic type constraint deletion) deletes one of constraints of
generics. If only one constraint exits the whole clause where is deleted.

Original code                          Mutated code with GCD
public class GenericTree<T>    public class GenericTree<T>
    where T: IComparable {               
        }                                    

The GCD operator can be applied only after checking that deletion of the constraint
would not cause invalid operation of T type objects. For example, in methods of class
GenericTree methods of interface ICloneable.CompareTo could be called. A relation
between generic types should also be checked in order to prevent inconsistencies of
constrains for a common type T. Therefore the operator is difficult to implement.

The GCI operator (Generic type constraint insertion) adds in generics a constraint
applied for one of parameters. The mutated code will be valid only if all generic in-
stances satisfy the new constraint.

Original code                          Mutated code with GCI
public class GenericTree<T>    public class GenericTree<T>
    where T: IComparable {               
        }                                    

The GOC operator (Generic type parameter order change) changes the order of pa-
rameters of a generic type. In instances of the generic type all variables of type T would
be of type R and vice versa. The operator mimics a programmers’ mistake of wrong
parameters order. The operator should be practically used if considered parameters have
no constraints or have the same constraints, because the code cannot compile when pa-
rameters have constraints that are violated after change of parameters.

Original code                          Mutated code with GOC
public class GenericTree<T, R>    public class GenericTree<R, T>
    where T: IComparable {               
        }                                    

The GCB operator (Generic type required class change to base class) changes a class
required in a constraint of a type into its base class. In the below example class Button
inherits from class Control. The operator can be only applied if the members of the
original derived class are not used in the generic type and if relations between generics
are not violated.

Original code                          Mutated code with GCB
public class GenericTree<T>    public class GenericTree<T>
    where T: Button {               
        }                                    

Using the GCC operator (Generic type required class change to derived type), the
class specified in a constraint is substituted by its derived class. Before this mutation a
set of classes for which generic instances are created should be verified.

Original code                          Mutated code with GCC
public class GenericTree<T>    public class GenericTree<T>
    where T: Control {               
        }                                    

The following mutation operators were proposed for the generic types.

The GCD operator (Generic type constraint deletion) deletes one of constraints of
generics. If only one constraint exits the whole clause where is deleted.
Nullables

C# 2.0 presented a new concept called nullable types to initialize value types to null values, e.g. values in an undefined state. Value types are primitive types such as numbers. A question mark (?) suffix following a data type designator specifies that the type can accept null values. Two mutation operators are proposed.

The NNU operator (Type reference change from nullable into the underlying type) can be applied only if properties Value, HasValues, operator ? and assignment to null value were not used in the context of the considered variable.

```
Original code          Mutated code with NNU
int ? i;               int i;
```

Correct application of the NUN operator (Type reference change from value into nullable), as well as NNU, could generate equivalent mutants, if original and mutated programs handles the variable in the same way.

```
Original code          Mutated code with NUN
int i;                 int ? i;
```

Static class

A class can be declared static, indicating that it contains only static members and no instance of the class can be created.

Usage of the SKD operator (Static keyword deletion) makes possible to create an instance of the class. If only one change per mutant is assumed the mutant would be equivalent to the original program.

```
Original code          Mutated code with SKD
static class ....      class ....
```

Partial classes and partial methods

Partial classes were proposed in C# 2.0. Definition of a class, a struct or an interface can be split over two or more source files. A source file contains a type or method definition, and all parts are combined when the application is compiled. Omitting a keyword partial for a class is not a valid mutation as it would be detected at compile time.

Application of the keyword partial was extended in C# 3.0. A partial class or struct can contain a partial method. A partial method declaration consists of two parts: its signature and its implementation. The signature is delivered in one part of the class. An implementation can be defined in the same or another part, optionally. If the implementation is missing the signature and all calls to this method are removed at compile time.

The mutation operator PRD (Partial method implementation deletion) deletes an implementation of a partial method from C# source code. The modified code would compile properly but the functionality of the expected method would be missing and this fact should be detected by adequate test cases.

```
Original code
partial class ClassPart {.... //Definition in file ClassPart1.cs
    partial void MyPartialMethod(string arg);
}
partial class ClassPart {... //Implementation in file ClassPart2.cs
    partial void MyPartialMethod(string arg) { //method body }
}
In the Intermediate Language a partial method is no more indistinguishable. At compile time both parts of partial methods are merged or the method is removed if the implementation is missing. Partial methods are further implemented as any other methods.

**Auto-implemented (automatic) properties**

Properties are used in C# to read, write, or compute the values of private fields. They are in fact special methods called assessors. Properties were treated in some mutation operators similarly as other class members, for example: JSC (static modifier change), MCO (member call from another object). Special operators were devoted to get and set methods, like EAM (accessor modifier change) and EMM (modifier method change). Taking into account special features of C# language, mutation operators dedicated to properties were also proposed and experimentally evaluated - PRM - property replacement with member field, and OPD - overriding property deletion [5].

In C# 3.0 declarations of properties were extended. Auto-implemented properties have no explicit declaration of a private field and no specification of simple get and set operations. For such properties the mutation operator PRM will be never applicable, because a property cannot be called by mistake instead of its field or vice versa. The mutation operator OPD deals with omission of a property that forces the usage of the appropriate property from a base class. It can be applicable only if a class does not inherit from an abstract type. This mutation operator can be used in the same way for regular and automatic properties. No new operators are proposed.

```csharp
public int Counter {get; set;}      //automatic property
```

An automatic property is converted by the compiler to a standard form of a regular property. It is invisible in the Intermediate Language and treated as any other property.

**Object Initializers**

Object initializer is a construction introduced in C# 3.0 for initialization of public properties and fields of classes. It can be completed in a simpler and quicker way than using a standard initialization notation. Object initializers are often used with automatic properties. Their syntax is strictly checked by the compiler. A possible mistake would be an initialization with wrong values. Such mutation would correspond for example to a standard mutation of constants, similar to CRP - constant replacement.

The code of object initialization would be translated into explicit assignment of each property. Therefore in the Intermediate Language object initializers are not visible.

**Implicitly typed local variables**

Local variables can be created in C# 3.0 without an explicit defined type. A variable is initialized with a certain expression in its definition. A type of this expression is counted also as the type of the variable. A keyword `var` is used to denote such variable.

```csharp
var flag = true;    //flag is boolean
var tab = new[] { 0, 1, 2, 3 }; //tab is int[]
```
Omission of the `var` keyword or omission of the initialization part would be detected at compilation time. Association of an incorrect type can be caused by initialization of another expression. This kind of a mistake could be mimic by standard mutation referring to constants, expression operators (if any), e.g. CRP constant replacement.

Intermediate code generated from a such variable is the same as the code created for an analogous variable with its type explicitly given and initialized with the same value. Hence, this construction cannot be mutated in the Intermediate Language.

**Anonymous types**

Anonymous types are used in C# 3.0 for fast defining and applying a type. The type represents some states. Only its properties would be defined. Anonymous types use two above mentioned constructions: implicitly typed local variables and object initializers.

```csharp
var me = new { Name = "John", Surname = "Smith"};
Console.WriteLine("My name is {0} {1}.", me.Name, me.Surname);
```

For an anonymous type a class will be created that inherits directly from a standard `System.Object`. A generated constructor takes parameters equal to values of initializations in the anonymous type. The class would be not visible at the C# language level. Also in this case the syntax is strongly controlled, and basic mutations could refer to different or swapped values of initialization.

Similarly, as for implicitly typed local variables and object initializers, application of anonymous type can be not mutated in the Intermediate Language, because this entire construction is transformed into a standard class.

**Extension methods**

An existing type (e.g. a class) after compilation had a defined interface. Changing of its interface was not possible, unless the source code was edited and recompiled, or a new derived type was created. For example, we could not add a new method to a class without having an access to its source code. In C# 3.0 a new construction of extension method was introduced enabling extending an interface at the C# language level. An extension method is implemented as a static method. The first parameter of the extension method defines an extended type. The parameter is proceeded by the keyword `this`.

```csharp
public static class MyExtensionClass {
    public static string MyMethod(this string arg) {
        s[0] = arg[0];
        return new string (s);
    }
}
```

An extension method can be called as if it were a new instance method of the extended type, or as a static method of the class in which it was declared. In the later case an object of the extended class is passed as a parameter to the method. This construction would be violated after omitting the keyword `this` in the declaration of an extension method and obtaining a valid static method. However such error would be detected when the method is called and therefore it would be not a proper mutation.

In the intermediate level an extension method is visible as a static method of a static class. It is denoted only with the attribute `ExtensionAttribute`. Consequently no special mutation operator would be applicable for this mechanism.
Anonymous methods, Covariance and contravariance in delegates

Delegates are one of very important programming constructions of C#. Functionally they can perform tasks similar as pointers to functions in C/C++ languages.

But delegates are strongly typed and can point only at a method that signature is consistent with its signature, i.e. a return type, number and types of parameters are consistent. In earlier versions of C# a delegate has to be declared as a named method. Since C# 2.0 anonymous methods can be used for defining delegates in a place of its assignment. Using covariance and contravariance a method passed to a delegate is less restrictive in its return type and parameters.

For delegates three mutation operators were proposed and evaluated experimentally [6]. The most promising one was the DMC operator – delegate method change. It can be applied for standard delegates, and delegates in a notation of anonymous methods, as well as extended to the rules of covariance and contravariance.

Lambda expression

In C# 3.0 a new construction so-called lambda expression can be used. It can be applied where a strongly typed delegation or anonymous method can be used. Lambda expressions use the lambda operator "=>", which is characteristic to the lambda calculus. The left side of the operator specifies its input parameters (if any) and the right side holds the expression or statement block processing the parameters.

```
List<int> list = new List<int> {1, 2, 3, 4, 5}; //simple list
List<int> foundedList = list.FindAll(i => i>2); //lambda expression
```

Mutation of input parameters or an output expression can be realized by standard mutation, e.g. substitution of one variable by another one, mutation of relation operators (ROR relational operator replacement), etc. A mistake of a lambda operator, like using of "=" in place of "=>", would be detected by the compiler. Also omitting of the whole lambda expression would be recognized if it is expected as a method parameter.

Lambda expressions are converted by the compiler to definitions of anonymous methods. They are in practice only a more compact notation for describing such methods. Therefore they are not distinguishable at the Intermediate Language level.

LINQ (Language Integrated Query)

Many applications require interaction with sources of various data, for example relational data bases, XML documents, collections like tables, lists, queues. Unifying data access, an integrated query language LINQ was introduced in C# 3.0. It provides clauses (from, where, select, ...) that allow to access different data in a similar way as using SQL. At compile time operators of C# LINQ are converted into appropriate calls of extension methods. Their parameters are specified as lambda expressions (as delegates).

The syntax of LINQ is very similar to that of SQL. Therefore the mutation operators proposed for SQL queries could also be used in C#, similarly to mutation categories: SC - mutations of SQL clauses, NL – mutations of NULL values, IR - replacement of identifiers in the clauses [27]. The details are beyond the scope of this chapter.

Apart from the constructions discussed in this subsection, mutation of other features introduced in C# was also considered. Details about several other possible mutation...
operators are omitted, as for example, swapping of an iterator in a `foreach` loop in classes with many iterators of the same kind, changing of a namespace alias qualifier, restriction of an accessibility level of set accessors on properties and indexers. According to examination, and/or experimental evaluation, either these situations deliver valid mutants but encounter very rarely or mainly equivalent mutants would be generated.

Table 1. Mutation of new C# 2.0 and 3.0 features

<table>
<thead>
<tr>
<th>C# feature</th>
<th>Abbreviation</th>
<th>Mutation operator</th>
<th>Application in the Intermediate Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generics (2.0)</td>
<td>GCD</td>
<td>Generic type constraint deletion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GCI</td>
<td>Generic type constraint insertion</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GOC</td>
<td>Generic type parameter order change</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GCB</td>
<td>Generic type required class change to base type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GCC</td>
<td>Generic type required class change to derived type</td>
<td></td>
</tr>
<tr>
<td>Nullable (2.0)</td>
<td>NNU</td>
<td>Type reference change from nullable into the underlying type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NUN</td>
<td>Type reference change from the ordinary value type into nullable</td>
<td></td>
</tr>
<tr>
<td>Static class (2.0)</td>
<td>SKD</td>
<td>Static keyword deletion</td>
<td></td>
</tr>
<tr>
<td>Partial classes (2.0)</td>
<td>PRD</td>
<td>Partial method implementation deletion</td>
<td>No</td>
</tr>
<tr>
<td>Automatic properties (3.0)</td>
<td>OPD</td>
<td>Overriding property deletion</td>
<td>No</td>
</tr>
<tr>
<td>Object initializers (3.0)</td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Implicitly typed local variables (3.0)</td>
<td></td>
<td>Standard operators for initialization expression</td>
<td>No</td>
</tr>
<tr>
<td>Anonymous types (3.0)</td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Extension methods (3.0)</td>
<td></td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>Anonymous methods Covariance and contravariance (2.0)</td>
<td>DMC</td>
<td>Delegate method change (extended)</td>
<td></td>
</tr>
<tr>
<td>Lambda expression (3.0)</td>
<td></td>
<td>Standard operators for parameters and expression used in lambda expression</td>
<td>No</td>
</tr>
<tr>
<td>Language Integrated Query (3.0)</td>
<td></td>
<td>Mutation operators similar to mutations of SQL language</td>
<td></td>
</tr>
</tbody>
</table>

4. EXPERIMENTS

Evaluation of mutation operators dealing with specific language features should be realized with programs in which such constructions are really used, and not only inten-
tionally added for experimental purposes. The problem is that some time has to elapse before programmers get customize to the features since the introduction of them into the programming language.

The programs used in experiments are listed in Tab. 2. They were developed independently of the mutation experiments and included considered constructions of C# 2.0. Mutation operators of generics, nullable and static classes were applied by hand, therefore no quantitative results but only qualitative observations will be provided.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClientServerPerl</td>
<td>3941</td>
<td>a student project</td>
</tr>
<tr>
<td>Phantom</td>
<td>114</td>
<td><a href="http://www.itu.dk/~sestoft/gcssharp/Phantom.cs">http://www.itu.dk/~sestoft/gcssharp/Phantom.cs</a></td>
</tr>
</tbody>
</table>

Application of the GCD operator requires checking of restrictions in usage of a constraint to be deleted in a generic type. But, a deliberate usage of a constraint in a generic type is often associated with functions that prevent from mutation of the constraint. In the examined programs generic types were exhaustively used and the GCD operator was not applicable. It might be more helpful in programs at earlier stages of the development or written by programmers less experienced in the application of generic types.

Adding a new constraint in generics with the GCI operator required a careful examination of related generic instances. In all cases the operator generated valid mutants, no differences in the program behaviour were observed. After program examination the mutants were counted as equivalent ones, which could not be killed by any test case.

The GOC operator changes the order of parameters of generic types if their types are consistent. However, in evaluated programs in the most cases the parameters have different types and could not be swapped. Only in one class the operator could be applied. The parameters not only had the same types but also analogous behaviour, and the resulting mutant was an equivalent one.

The remaining two operators of generics, GCB and GCC require strong conditions on derived methods used in a generic type. In non of generics used in the experiments such operators could be applied. It can be assumed that only in case of generics combined with an extended inheritance hierarchy the operators would be useful.

The problems with the operator NNU were similar to GCD. A purposeful declaration of a nullable was associated with the usage of methods and properties `GetValueOrDefault()`, `Value`, `HasValue`. It prevented application of the NNU operator. It might be useful in a program with the premature usage of a nullable. In the case of the second operator dealing with nullable – NUN, all generated mutants were equivalent.

Finally, the SKD operator converted a `static` class into a regular one. Its application was not very frequent and resulted in equivalent mutants, i.e. no test case could fail.
5. CONCLUSIONS

The main goal of the work was recognition whether the programming features introduced in C# 2.0 and 3.0 could be handled using a mutation approach. The potential dedicated mutation operators would be used for verification of test cases that should detect faults in such constructions.

The general results are not promising and the operators were not incorporated to the CREAM and ILMutator tools. Restrictions on the type usage and no flexibility in the construction syntax make simple programming mistakes that change considered constructions less probable than those known from standard mutations or even object-oriented ones.

Several emerging features were recognized inappropriate for mutation operators. They required careful specification in order to avoid creation of invalid mutants. Therefore, only in special cases the mutants could be generated. Those created mutants were also often equivalent ones, and as such could be not used for evaluation of test cases.

Most of the features introduced in C# 3.0 are only a shortened syntax of constructions known from the previous versions. Therefore, they can be not distinguishable in the Intermediate Language and can be not mutated at this level, which is more effective for introducing mutations. On the other hand a whole new mechanism of C# 3.0 - LINQ (Language Integrated Query) could be analysed with the mutation operators similarly to SQL programs. Programs using discussed features can be of course tested with typical functional tests and the white box tests should take into account such constructions.

REFERENCES